The Working of AX -- an Automaton eXtractor from C code

Included in this note are three small examples that illustrate the model extraction method from ANSI C code with the tool Ax.
We discuss:

- A simple implementation of the alternating bit protocol in ANSI C.
- Two different implementations of a standard quicksort algorithm. The first implementation is the version printed verbatim in Kernighan & Pike's book "*The Practice of Programming*" (Addison-Wesley, 1999) on page 33. The second implementation is the same algorithm rewritten as a non-recursive procedure.

Example 1: Alternating bit protocol.

The source text for this program, written in standard ANSI C, is as follows. The line numbers in the left margin were added for easier reference.

```
1  #include <stdio.h>
2
3  /*
4   * C version of alternating bit protocol
5   * to show that the code need not be
6   * structured as a state machine, as
7   * with the @ format
8   * GJH 3/6/2000
9   */
10 typedef char uchar;
11
12 typedef struct Buffer {
13     int size;     /* current size of buffer */
14     uchar *cont; /* buffer contents */
15 } Buffer;
16
17 extern int get_data(Buffer *);
18 extern int put_data(Buffer *);
19
20 int
21 abp_sender(int N)
22 {   Buffer Bufinp, Bufout;
23     short  s, S=0, cnt=0;
24
25     Bufout.size = 1;
26     Bufout.cont = "M";
27     while (cnt++ < N)
28     {       if (!get_data(&Bufout))
29                   break;
30             send(&Bufout, S);
31             recv(&Bufinp, &s);
32             if (*Bufinp == 'A' && s == S)
33                   S = 1 - S;
34     }
35     return cnt;
36 }
37
38 int
39 abp_receiver(void)
40 {   Buffer  Bufinp, Bufout;
41     short  s, E=0, cnt=0;
42
43     Bufout.size = 1;
44     Bufout.cont = "A";
45     while (recv(&Bufinp, &s))
```

```
46      {       cnt++;
47              send(&Bufout, s);
48              if (s == E)
49              {       if (!put_data(&Bufinp))
50                              break;
51                      E = 1 - E;
52      }       }
53      return cnt;
54 }
```

This C program defines the behavior of a sender and of a receiver process. To run the protocol one can instantiate two independent processes (asynchronous threads of execution): one process to execute the sender's code and one process to execute the receiver's code. Independently of the part of the implementation shown here, one then provides two external routines, **get_data()** obtains the data to be transmitted at the sender side, and **put_data()** delivers data to its ultimate destination at the receiver side. The details of the code are not of primary interest here, but the process of converting it automatically into an abstract model, guided by a user defined conversion (lookup) table.

Just using this program as input, and without a first version of a lookup table just yet, we can extract a first version of a verification model in Promela. The tool 'ax' will generate a default lookup table for us as well, that we can then edit to adjust the abstraction that is applied.
We extract the two parts of the model separately: a part for the receiver and a part for the sender. The two parts are combined in a hand-written wrapper that we place around the code, as shown below.
The first version of the two parts of the model is generated as follows with the commands:

```
$ ax -a abp_receiver abp.c
$ ax -a abp_sender  abp.c
```

The two parts of the model are extracted into the files **"abp_receiver.spn"** and **"abp_sender.spn"**, and the two default lookup tables are written into the files **"abp_receiver.lut"** and **"abp_sender.lut"**. The contents of these two files after this first pass is as follows:

1. Contents of file abp_receiver.lut:

```
# Ax Lookup Table abp_receiver.lut
D: Buffer Bufinp,Bufout;            hide
D: short s,E=0,cnt=0;               hide
A: Bufout.size=1                    hide
A: Bufout.cont="A"                  hide
C: recv(&(Bufinp),&(s))             true
C: !recv(&(Bufinp),&(s))            true
cnt++                               print
F: send(&(Bufout),s)                print
C: (s==E)                           true
C: !(s==E)                          true
C: (!put_data(&(Bufinp)))           true
C: !(!put_data(&(Bufinp)))          true
A: E=(1-E)                          hide
R: return                           print
```

2. Contents of the file abp_sender.lut:

```
# Ax Lookup Table abp_sender.lut
D: Buffer Bufinp,Bufout;            hide
D: short s,S=0,cnt=0;               hide
A: Bufout.size=1                    hide
A: Bufout.cont="M"                  hide
C: (cnt++<N)                        true
C: !(cnt++<N)                       true
C: (!get_data(&(Bufout)))           true
```

```
C: !(!get_data(&(Bufout)))          true
F: send(&(Bufout),S)                print
F: recv(&(Bufinp),&(s))             print
C: ((*Bufinp=='A')&&(s==S))         true
C: !((*Bufinp=='A')&&(s==S))        true
A: S=(1-S)                          hide
R: return                           print
```

Ax assigns default mappings to each basic statement that it encounters. Where the context is unambiguous, it will classify the statements as either a declaration (prefix "D:"), a condition (prefix "C:"), an assignment (prefix "A:"), a function call (prefix "F:"), a return statement (prefix "R:"). The prefix is just optional decoration of the lookup table entries. If it cannot be determined from context (as in the case of the "cnt++" statements) it is omitted. All prefixes could be omitted without loss of functionality.
The default mappings used here are "true" for conditions (and their negations), "print" for function calls and return statements, "hide" for declarations and assignments. The defaults are just meant as a starting point for the definition of a proper abstraction by the user. To do so, we edit these default lookup tables to make them reflect more accurately what aspect of the implementation we are interested in and what we want to abstract from. Sample revised tables that capture a possible abstraction are as follows.

Contents of the file abp_receiver.lut:

```
# Ax Lookup Table abp_receiver.lut
D: Buffer Bufinp,Bufout;    hide
D: short s,E=0,cnt=0;       byte E, s
C: recv(&(Bufinp),&(s))     rq?s
C: !recv(&(Bufinp),&(s))    false
cnt++                       hide
C: (s==E)                   keep
C: !(s==E)                  keep
C: (!put_data(&(Bufinp)))   false
C: !(!put_data(&(Bufinp)))  print
A: Bufout.size=1            hide
A: Bufout.cont="A"          hide
A: E=(1-E)                  keep
F: send(&(Bufout),s)        sq!s
R: return                   hide
```

Contents of the file abp_sender.lut:

```
# Ax Lookup Table abp_sender.lut
D: Buffer Bufinp,Bufout;    hide
D: short s,S=0,cnt=0;       byte s, S
C: (cnt++<N)                true
C: !(cnt++<N)               false
C: (!get_data(&(Bufout)))   false
C: !(!get_data(&(Bufout)))  print
F: send(&(Bufout),S)        rq!S
C: ((*Bufinp=='A')&&(s==S))s==S
C: !((*Bufinp=='A')&&(s==S))else
A: Bufout.size=1            hide
A: Bufout.cont="M"          hide
A: S=(1-S)                  keep
F: recv(&(Bufinp),&(s))     sq?s
R: return                   hide
```

By repeating the two commands from above with the new, now explicitly defined lookup tables, we generate the final version of the verification models.

```
$ ax -a abp_receiver abp.c
$ ax -a abp_sender abp.c
```

This gives the following result. The lookup table files are not modified by Ax in this second pass, since they completely cover all basic statements used in the source. In the two displays below we've made some cosmetic adjustments to the printed form, e.g., by indenting some comments. The two files are machine generated by Ax.

Contents of file abp_receiver.spn:

```
active proctype abp_receiver()
{       /* 40: D: Buffer Bufinp,Bufout;  */
  byte E, s; /* 41: D: short s,E=0,cnt=0;  */
        /* 43: A: Bufout.size=1 */
        /* 44: A: Bufout.cont="A" */
L1:
  do
  :: rq?s;    /* C: recv(&(Bufinp),&(s)) */
       /* cnt++ */
      sq!s;  /* 47: F: send(&(Bufout),s) */
      if
      :: (s==E);
          printf("C: !(!put_data(&(Bufinp)))\n");
          E=(1-E);  /* line 51 */
      ::  !(s==E);
      fi;
  od;
  /* 53: R: return */
}
```

Contents of file abp_sender.spn:

```
active proctype abp_sender()
{       /* 22: D: Buffer Bufinp,Bufout;  */
  byte s, S; /* 23: D: short s,S=0,cnt=0;  */
        /* 25: A: Bufout.size=1 */
        /* 26: A: Bufout.cont="M" */
L0:
  do
  ::  printf("C: !(!get_data(&(Bufout)))\n");
      rq!S;   /* 30: F: send(&(Bufout),S) */
      sq?s;   /* 31: F: recv(&(Bufinp),&(s)) */
      if
      :: s==S;       /* C: ((*Bufinp=='A')&&(s==S)) */
           S=(1-S); /* line 33 */
      :: else;       /* C: !((*Bufinp=='A')&&(s==S)) */
      fi;
  od;
  /* 35: R: return */
}
```

If we edit the source file abp.c, we can reuse the lookup tables from above to re-extract abstract models from the modified C code. If new statements were introduced, the model extractor will add default entries for them in the lookup table and warn the user about their presence, so that they can be adjusted to conform to the abstraction focus that was chosen. If statements were omitted, the model extractor will comment them out of the lookup table by placing a comment symbol (#) at the start of these entries. For even significant revisions of the source, taking days for the programmer to make, an update of the lookup tables to bring them back in sync with the new version of the code typically takes no more than a few minutes of user time. (The alternative of rebuilding a complete verification model for the new source by hand would more likely take days – approaching the investment of time that the programmer made.)

We can inspect the behavior of the abstracted implementation with the logic model checker Spin. First we join the two parts of the model in a simple Promela wrapper that defines minimal context for the two

processes. The wrapper below defines two abstract channels via which the processes can exchange their messages, and includes the text of the two processes. The text of this wrapper, stored in a file called **abp**, is:

```
chan rq = [1] of { byte };
chan sq = [1] of { byte };

#include "abp_receiver.spn"
#include "abp_sender.spn"
```

Now we can run spin on this file. First, we can look at the first 20 steps in a simulation run, looking only at message exchanges:

```
$ spin -c abp | sed 20q
proc 0 = abp_receiver
proc 1 = abp_sender
        C: !(!get_data(&(Bufout)))
q\p   0   1
  1   .   rq!0
  1   rq?0
  2   sq!0
  2   .   sq?0
C: !(!put_data(&(Bufinp)))
        C: !(!get_data(&(Bufout)))
  1   .   rq!1
  1   rq?1
  2   sq!1
  2   .   sq?1
C: !(!put_data(&(Bufinp)))
        C: !(!get_data(&(Bufout)))
  1   .   rq!0
  1   rq?0
  2   sq!0
C: !(!put_data(&(Bufinp)))
```

This shows the two processes exchanging the sequence numbers and correctly retrieving and depositing data during the run. A verification run can be more illuminating, checking the system for possible deadlocks, and answering any other logical query that the user can formulate about the operation of the system.

```
$ spin -a abp
$ cc -o pan pan.c
$ pan
(Spin Version 3.3.10 -- 6 March 2000)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates   +

State-vector 36 byte, depth reached 13, errors: 0
       14 states, stored
        2 states, matched
       16 transitions (= stored+matched)
        0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493  memory usage (Mbyte)
```

```
unreached in proctype abp_receiver
        (0 of 12 states)
unreached in proctype abp_sender
        (0 of 12 states)
```

This verification run proves, for instance, absence of deadlock for the alternating bit protocol.

Example 2: The Quicksort algorithm.

The next example serves to show that the model extraction can be applied to arbitrary ANSI C code as input, without structuring or style requirements. The code need not be written in a format resembling a traditional finite state machine: the model extractor can generate the state machine structure from the control flow skeleton of the program for any C program. Model checking for the quicksort algorithm itself is not a very fruitful exercise, though, since the algorithm involves no concurrency or process interaction, so this is just an example of model extraction, not of model checking. For sequential and deterministic algorithms of this type it is generally better to analyze their properties with more conventional techniques. Here first is the code from the quicksort algorithm as it appears in a recent textbook (Kernighan & Pike's "*The Practice of Programming*" (Addison-Wesley, 1999), page 33).

```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
        int temp;

        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}

/* quicksort: sort v[0]..v[n-1] into increasing order */
void quicksort(int v[], int n)
{
        int i, last;

        if (n <= 1)  /* nothing to do */
                return;

        swap(v, 0, rand() % n);    /* move pivot */
        last = 0;                  /*   to v[left] */
        for (i = 1; i < n; i++)    /* partition */
                if (v[i] < v[0])
                        swap(v, ++last, i);

        swap(v, 0, last);     /* restore pivot */
        quicksort(v, last);   /* recursively sort */
        quicksort(v+last+1, n-last-1); /*   each half */
}
```

We can apply the model extractor to this code, as is, and generate a default lookup table that we can then edit as before. The default lookup table that is generated for the quicksort routine looks as follows.

Contents of quicksort.lut:

```
# Ax Lookup Table quicksort.lut
D: int i,last;              hide
C: (n<=1)                   true
C: !(n<=1)                  true
R: return                   print
F: swap(v,0,(rand()%n))     print
```

```
A: last=0                    hide
A: i=1                       hide
C: (i<n)                     true
C: !(i<n)                    true
C: (v[i]<v[0])               true
C: !(v[i]<v[0])              true
F: swap(v,++last,i)          print
i++                          print
F: swap(v,0,last)            print
F: quicksort(v,last)         print
F: quicksort(((v+last)+1),((n-last)-1))        print
```

Note that the routine is recursive, and in the default lookup table the recursion is mapped away. We can edit the lookup table to define the required abstraction, retaining all the functionality from the original program, including the recursive calls, but redefined to a smaller domain.

Contents of revised quicksort.lut:

```
# Ax Lookup Table quicksort.lut
D: int i,last;          short i, last, temp;
C: (i<n)                keep
C: !(i<n)               keep
i++                     keep
C: (n<=1)               (n-a<=1)
C: !(n<=1)              else
C: (v[i]<v[0])          (v[i]<v[a])
C: !(v[i]<v[0])         else
A: last=0               last=a
A: i=1                  i=a+1
F: swap(v,0,(rand()%n))         temp = v[a]; v[a] = v[(n+a)/2]; v[(n+a)/2] = temp
F: swap(v,++last,i)             last++; temp = v[i]; v[i] = v[last]; v[last] = temp
F: swap(v,0,last)               temp = v[a]; v[a] = v[last]; v[last] = temp
F: quicksort(v,last)            run quicksort(a,last); _nr_pr == _pid+1
F: quicksort(((v+last)+1),((n-last)-1))       run quicksort(last+1, n); _nr_pr == _pid+1
R: return               goto done
```

We repeat the model extraction with the edited table and obtain the following abstract model from the code. We extract the model without the proctype enclosure, using the option "-n" instead of the earlier "-a". (Ax has options for generating the models as fully instantiated processes (-a), as uninstantiated process declarations (-p) or as process bodies (-n), as used in this case.)

```
$ ax -n quicksort qsort.c
```

Contents of quicksort.spn:

```
  short i, last, temp;      /* 18: D: int i,last;  */
  if
  :: (n-a<=1);              /* C: (n<=1) */
     goto done;             /* 21: R: return */
  :: else;                  /* C: !(n<=1) */
  fi;
  temp = v[a]; v[a] = v[(n+a)/2]; v[(n+a)/2] = temp;
                /* 23: F: swap(v,0,(rand()%n)) */
  last=a;       /* 24: A: last=0 */
  i=a+1;        /* 25: A: i=1 */
L0:
  do
  ::  (i<n);
      if
      :: (v[i]<v[a]);           /* C: (v[i]<v[0]) */
         last++; temp = v[i]; v[i] = v[last]; v[last] = temp;
                /* 27: F: swap(v,++last,i) */
      :: else;                  /* C: !(v[i]<v[0]) */
      fi;
```

```
        i++;
    ::  !(i<n);  -> break
    od;
    temp = v[a]; v[a] = v[last]; v[last] = temp; /* 29: F: swap(v,0,last) */
    run quicksort(a,last); _nr_pr == _pid+1;     /* 30: F: quicksort(v,last) */
    run quicksort(last+1, n); _nr_pr == _pid+1;
  /* 31: F: quicksort(((v+last)+1),((n-last)-1)) */
goto done;    /* 31: R: return */
```

We now have to define the wrapper that defines the context for the generated model. In the wrapper we can pass some data to be sorted to the quicksort process, and check that it was sorted correctly after that process completes. For the example we'll use the following wrapper for the quicksort model:

Contents of file 'quick':

```
short v[16];

inline checkresult() {
        d_step {
        i = 0;
        do
        :: i < n-1 ->
                assert(v[i] <= v[i+1]);
                i++
        :: else ->
                break
        od;
        skip
        }
}

proctype quicksort(short a, n)
{       atomic {
        printf("quicksort %d - %d\n", a, n);
#include "quicksort.spn"
done:   skip
        }
}

init {
        short i, n;
        if
        :: n = 6        /* even value for n */
        :: n = 5        /* odd value for n */
        :: n = 0        /* boundary case */
        :: n = 1        /* boundary case */
        fi;
        do
        :: i < n -> i++;
                if
                :: v[i] = 15
                :: v[i] = 9
                :: v[i] = 4
                :: v[i] = 27
                :: v[i] = 11
                fi;
        :: else ->
                break
        od;
        run quicksort(0,n);
        _nr_pr == _pid+1;
        checkresult()
}
```

We seed the model with some random input date. We chose between 0 and 6 numbers, arbitrarily from the set 4, 9, 11, 15, and 27 and put them in random order (using non-deterministic choices that are native to the modeling language Promela). Then we call the quicksort routine over the range of numbers selected. The line "`_nr_pr == _pid+1`" in Promela guarantees that the last procedure call (really a process instantiation) completes before we continue. The final call to "`checkresult()`" will verify that the numbers were correctly placed in numerical order, whatever their initial order may have been.

Example 3: A Non-Recursive Version of Quicksort:

Although the model extraction is straightforward for the recursive version of the quicksort algorithm, the definition of the lookup table requires some thought. It can be much simpler if the original C code is closer in nature to the target modeling language (in this case Promela). As an example of this, below is a revised version of the quicksort algorithm, functionally equivalent to the original, not relying on recursion.

Revised qsort.c:

```
#include <stdio.h>
#include <stdlib.h>

#define MaxI        100000
#define MaxD        64

int debug = 0;

void
assert(int e)
{
        if (!e)
        {       printf("stack overflow\n");
                exit(1);
        }
}

#define swap(i,j)   temp = v[i]; v[i] = v[j]; v[j] = temp
#define putQ(x,y)   assert(Qsz < MaxD); from[Qin] = x; \
                    upto[Qin] = y; Qin = (Qin+1)%MaxD; Qsz++
#define getQ()      a = from[Qout]; b = upto[Qout]; Qout = (Qout+1)%MaxD; Qsz--

void
main(void)
{       int i, n, j, k, last, temp;
        int v[MaxI], from[MaxD], upto[MaxD];
        int Qin=0, Qout=0, Qsz=0, a, b;

        for (n = 0; n < MaxI; n++)
                if (fscanf(stdin, "%d", &v[n]) == EOF)
                        break;

        if (n == 0) goto done;

        putQ(0, n-1);
        while (Qsz > 0)
        {       getQ();

                if (b-a <= 0)
                        continue;
                k = a+rand()%(b-a);
                swap(a, k);
                last = a;
                for (i = a+1; i <= b; i++)
```

```
                if (v[i] < v[a])
                {       last++;
                        swap(last, i);
                }

        swap(a, last);
        putQ(a, last);
        putQ(last+1, b);
    }

done:
    for (j = 0; j < n; j++)
        printf("%d\n", v[j]);
    exit(0);
}
```

The version shown above is a complete self-contained program, that reads in numbers from the standard input, sorts them with the quicksort algorithm, and then prints out the result on standard output.
The lookup table for the main routine can now be defined as follows, consisting mostly of **"keep"** mappings that are preserve the text from the source verbatim in the target:

Contents of main.lut:

```
# Ax Lookup Table main.lut

# map into smaller domain:

D: int v[100000],from[64],upto[64];     int v[8], from[8], upto[8]
F: assert((Qsz<64))                     assert((Qsz<8))
C: (n<100000)                           (n<8)
C: !(n<100000)                          !(n<8)
A: k=(a+(rand()%(b-a)))                 k=((a+b)/2)   /* determinize */
A: Qout=((Qout+1)%64)                   Qout=((Qout+1)%8)
A: Qin=((Qin+1)%64)                     Qin=((Qin+1)%8)

# literal, with syntactic conversion:

D: int i,n,j,k,last,temp;               int j,k,last,temp;
F: exit(0)                              hide
R: return                               checkresult()
C: (fscanf(&(__dj_stdin),"%d",&(v[n]))==(-1))  empty(qin)
C: !(fscanf(&(__dj_stdin),"%d",&(v[n]))==(-1)) qin?k; v[n] = k
Qsz++                                   Qsz++; assert(Qsz<=n+1)
Qsz--                                   Qsz--; assert(Qsz>=0)

# the rest is literal:

D: int Qin=0,Qout=0,Qsz=0,a,b;          keep
F: printf("%d\n",v[j])                  keep

C: (Qsz>0)                              keep
C: !(Qsz>0)                             keep
C: ((b-a)<=0)                           keep
C: !((b-a)<=0)                          keep
C: (i<=b)                               keep
C: !(i<=b)                              keep
C: (v[i]<v[a])                          keep
C: !(v[i]<v[a])                         keep
C: (j<n)                                keep
C: !(j<n)                               keep
C: (n==0)                               keep
C: !(n==0)                              keep
```

```
A: n=0                          keep
A: from[Qin]=0                  keep
A: upto[Qin]=(n-1)              keep
A: a=from[Qout]                 keep
A: b=upto[Qout]                 keep
A: temp=v[a]                    keep
A: v[a]=v[k]                    keep
A: v[k]=temp                    keep
A: last=a                       keep
A: i=(a+1)                      keep
A: temp=v[last]                 keep
A: v[last]=v[i]                 keep
A: v[i]=temp                    keep
A: v[a]=v[last]                 keep
A: v[last]=temp                 keep
A: from[Qin]=a                  keep
A: upto[Qin]=last               keep
A: from[Qin]=(last+1)           keep
A: upto[Qin]=b                  keep
A: j=0                          keep
i++                             keep
n++                             keep
j++                             keep
last++                          keep
```

The lookup table is used here mostly to restrict the maximum range of numbers and to convert the style of reading input and writing output. Finally, the wrapper for the extracted model in this context can be defined as follows:

```
chan qin = [6] of { int };

inline checkresult() {
        i = 0;
        do
        :: i < n-1 ->
                assert(v[i] <= v[i+1]);
                i++
        :: else ->
                break
        od
}

init {
        byte i, n;
        if
        :: n = 6        /* even value for n */
        :: n = 5        /* odd value for n */
        :: n = 0        /* boundary case */
        :: n = 1        /* boundary case */
        fi;
        do
        :: i < n -> i++;    /* placed together to get a merge pair */
                if
                :: qin!15   /* choose from n different or equal nrs */
                :: qin!9    /* we could also fill in the v[]'s directly */
                :: qin!4
                :: qin!27
                :: qin!11
                fi;
        :: else ->
                break
        od;
```

```
        skip;  /* syntax requirement */
        d_step {
#include "main.spn"
        }
}
```

As noted, the objective of this exercise is not to verify the quicksort algorithm, but to demonstrate that model extraction from arbitrary ANSI C code is possible with the Ax model extraction tool.